UNIVERSITY OF WISCONSIN
Computer Sciences Department

CS 537
Fall 2018

Barton Miller

Programming Assignment #1

Handed out: Tuesday, September 11
Due: Thursday, September 27 at 5pm

There are several goals for this assignment.

1. (Re-)familiarize yourself with the C programming language. Dealing with separate compilation is likely to be new to many of you.
2. Learn how to use some of the I/O facilities and library functions provided by UNIX and C. There are likely to be some new functions that you have not seen yet.
3. Get experience with some of the system level interfaces provided in Linux. Operating system have many interesting dark corners, and you will get a chance to peek around one of them.
4. Get some insights as to how an important Linux utility works.
5. Get experience with separate compilation, makefiles, and the Linux gdb debugger.

Your assignment is to write a simple version of the `ps` command. Your version of `ps`, called `537ps` be executed from a command line. It will read a variety of information about one or more running programs (processes) on the computer, and then display that information. As an example of what your program will do, consider a command such as:

```
537ps -p 1234 -s -U -c
```

This runs your `ps` command, and displays the status letter (in this case, running), amount of user time, and the command line for process 1234. You might expect the output to look like:

```
1234: R utime=150 [myprog -x -y file1 myoption]
```

UNIX/Linux Manual Pages and Other information Sources

Any serious systems programmer working on Linux (or other UNIX-like systems) needs to be familiar with the manuals and the online "man" facility. This will be a great help in working on these assignments. For example, if you wanted to know how the `ps` (list processes) command works, you would type:

```
man 1 ps
```

Or you can find the manual page on the web using Google. However, the versions installed on your particular Linux system, accessible via the `man` command, will be the definitive and accurate version for your system.

The UNIX manual is organized into many sections. You will be mainly interested in the first three sections. Section 1 is for commands, like `ls`, `gcc`, or `cat`. Section 2 is for UNIX

system calls (calls directly to the UNIX kernel), such as `fork`, `open`, or `read`. You will typically not use Section 2. The UNIX library routines are in Section 3. These are calls such as `atof`, or `strcpy`.

Another function in Section 3 that will be useful for this assignment is the C library function that handles command-line argument parsing. The man page for that function is found by typing:

```
man 3 getopt
```

Note that you want to make sure that you get the `getopt` from Section 3, as the `getopt` in Section 1 is a command, not a library (and not what you need). The cool part about the `getopt` manual page is that there is an "EXAMPLE" section that shows you how to use it. This example will help you to get started.

You will need to be able to read the contents of a Linux directory (in this case, `/proc`. For this purpose, you will use the `readdir` library function. The man page for that function is found by typing:

```
man 3 readdir
```

Here, the "3" is also essential, or you will get the man page for a lower-level function that is much harder to use. There are a lot of examples of code on the Web showing you how to use `readdir`. Here is one such example provided by IBM.

There is another, less well-known section of the manual, which will be critical for this assignment. You will need to understand the format of the `/proc` file system (sometimes called the `procfs`), and for that use, you would type:

```
man 5 proc
```

You can also learn about `/proc` from Web sources such as this: The proc File System. And here is a nice sample program that opens a directory and reads the directory entries.

More details about the online manual will be given in Discussion Section.

Program Features

Your program will will implement the features triggered by the following options.

`-p <pid>`
: Display process information only for the process whose number is `pid`. It does not matter if the specified process is owned by the current user. If this option is not present then display information for all processes of the current user (and only of the current user).

`-s`
: Display the single-character state information about the process. This information is found in the `stat` file in process's directory, looking at the third ("state") field. Note that the information that you read from the `stat` file is a character string. This option defaults to be **false**, so if it is not present, do not display this information. `-s-` is valid but has no effect.

`-U`
: Display the amount of user time consumed by this process. This information is found in the `stat` file in process's directory, looking at the "utime" field. This option

defaults to be **true**, so if it is not present, then this information is displayed. `-U-` turns this option off.

`-S`

Display the amount of system time consumed so far by this process. This information is found in the `stat` file in process's directory, looking at the "stime" field. This option defaults to be **false**, so if it is not present, then this information is **not** displayed. `-S-` is valid but has no effect.

`-v`

Display the amount of virtual memory currently being used (in pages) by this program. This information is found in the `statm` file in process's directory, looking at first ("size") field. This option defaults to be **false**, so if it is not present, then this information is **not** displayed. `-v-` is valid but has no effect.

`-c`

Display the command-line that started this program. This information is found in the `cmdline` file in process's directory. Be careful on this one, because this file contains a list of null (zero byte) terminated strings. This option defaults to be **true**, so if it is not present, then this information is displayed. `-c-` turns this option off.

Program Structure

Key to any good program, and a key to making your implementation life easier, is having a clean, modular design. With such a design, your code will be simpler, smaller, easier to debug, and (most important, perhaps) you will get full points.

Even though this is a simple program with few components, you will need to develop clean interfaces for each component. And, of course, each component will be in a separately compiled file, linked together as the final step. And, again of course, you will have a makefile that control the building (compiling and linking) of your code.

Some suggestions for modules in your design:

- *Options processing:* This module processes the command line options, setting state variables to record what the options specify.
- *Getting the process list:* If there is no `-p` option, then you will need to look through `/proc` to find the list of processes belonging to the user. This module will implement that functionality.
- *`stat` and `statm` parser:* This module will extract strings from the space-separated lists that are read from the `stat` and `statm` files.

Testing Your Program

First test the separate parts of your program, such as processing command line options, listing files in `/proc`, and reading and parsing the individual files in a process's `/proc` directory.

Next, start assembling these pieces into a whole program, testing the options one at a time. You will want to learn to use shell scripts, so you can set up sequences of command lines to run over and again. Make sure to try out an interesting variety of command. The TAs will provide you with suggestions what to test.

Deliverables

You will work in **groups of two** and turn in a single copy of your program, clearly labeled with the name and logins of both authors. Any exception to working in a group must be approved by Bart in advance; there must be some unusual reason for not working in a group.

You will turn in your programs, including all .c and .h files and your makefile. Also include a README file which describes a little bit about what you did for this project.

Note that you must run your programs on the Linux systems provided by the CS Department. You can access these machines from the labs on the first floor of the Computer Sciences Building or from anywhere on the Internet using the ssh remote login facility. If you do not have ssh on your Windows machine, you can download this client:

SSHSecureShellClient-3.2.9.exe

Handing in Your Assignment

Your CS537 handin directory is `~cs537-1/handin/your_login` where *your_login* is your CS login. Inside of that directory, you need to create a `proj1` subdirectory.

Copying your files to this directory is accomplished with the `cp` program, as follows:

```
shell% cp *.[ch] makefile README cs537-1/handin/your_login/proj1
```

You can hand files in multiple times, and later submissions will overwrite your previous ones. To check that your files have been handed in properly, you should list `~cs537-1/handin/your_login/proj1` and make sure that your files are there. The handin directories will be closed after the project is due.

As you are working in pairs, you should:

1. Submit only one copy of your code.
2. Make sure that both of your names are in comments at the top of each source file.
3. Both of you should create another file called partner.txt in **both** of your `proj1` directories, where you should have two lines that have each of CS login and netid of you and your partner.

Original Work

This assignment must be the original work of you and your project partner. Unless you have explicit permission from Bart, you may not include code from any other source or have anyone else write code for you.

Use of unattributed code is considered plagiarism and will result in academic misconduct proceedings (and "F" in the course and a notation on your transcript).

Last modified: Sun Sep 16 19:07:57 CDT 2018 by bart